WeeFence: Toward Making Fences Free in TSO *

Yuelu Duan, Abdullah Muzahid,† Josep Torrellas
University of Illinois at Urbana-Champaign
duan11@illinois.edu muzahid@cs.utsa.edu torrella@illinois.edu
http://iacoma.cs.uiuc.edu

ABSTRACT

Although fences are designed for low-overhead concurrency coordination, they can be expensive in current machines. If fences were largely free, faster fine-grained concurrent algorithms could be devised, and compilers could guarantee Sequential Consistency (SC) at little cost.

In this paper, we present *WeeFence* (or *WFence* for short), a fence that is very cheap because it allows post-fence accesses to skip it. Such accesses can typically complete and retire before the pre-fence writes have drained from the write buffer. Only when an incorrect reordering of accesses is about to happen, does the hardware stall to prevent it. In the paper, we present the WFence design for TSO, and compare it to a conventional fence with speculation for 8-processor multicore simulations. We run parallel kernels that contain explicit fences and parallel applications that do not. For the kernels, WFence eliminates nearly all of the fence stall, reducing the kernels' execution time by an average of 11%. For the applications, a conservative compiler algorithm places fences in the code to guarantee SC. In this case, on average, WFences reduce the resulting fence overhead from 38% of the applications' execution time to 2% (in a centralized WFence design), or from 36% to 5% (in a distributed WFence design).

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - Parallel Processors.

Keywords

Fences, Synchronization, Memory Consistency, Sequential Consistency, Parallel Programming, Shared-Memory Multiprocessors.

1. INTRODUCTION

Fences are instructions that programmers or compilers insert in the code to prevent the compiler or the hardware from reordering memory accesses [10, 25]. While there are different flavors of fences, the basic idea is that all of the accesses before the fence have to be finished (i.e., the loads have to be retired and the writes drained from the write buffer) before any access after the fence can be observed by any other processor. The goal is to prevent a reordering that could lead to an incorrect execution.

Fences are used for low-overhead concurrency coordination in places where conventional synchronization primitives such as locks would have too much overhead. In some cases, programmers insert explicit fences in algorithms with fine-grain sharing. For instance, this is the case in the Cilk THE [7] work-stealing algorithm. In the fast path of the algorithm, there are fences between two consecutive accesses to a queue (e.g., to *queue->head* and *queue->tail*, respectively) that, if reordered by the compiler or hardware, could cause incorrect execution.

In other cases, compilers insert fences. For example, in C++, the programmer can employ intentional data races for performance, and declare the relevant variables as *atomic* [2] (or *volatile* for Java). Such declaration prompts the compiler to insert a fence after the access and abstain from generating reordered code; the fence then prevents the hardware from reordering accesses dynamically.

Fences can be expensive in current machines. A simple test on a desktop with an 8-threaded Intel Xeon E5530 processor reveals that a fence introduces a significant visible overhead. If the write buffer is empty, the fence introduces about 20 cycles; if there are many pre-fence write misses, then it may take an order of magnitude more cycles until all the writes drain from the write buffer.

If fences did not stall the pipeline and, instead, had a negligible performance cost, software could take advantage in two ways. First, programmers could write faster fine-grained concurrent algorithms. Second, it would be feasible for C++ (or Java) programs to guarantee SC execution *at little performance cost*. To see why, recall that a C++ compiler is required to generate SC code as long as any data race accesses are on variables declared as atomic. If fences could be skipped while retaining correctness, programmers could declare all shared data as atomic, triggering the insertion of a fence after every single shared data access. However, hardware reordering would not be curtailed. Moreover, while there would be a performance overhead due to limiting compiler optimizations, recent work has indicated that such effect may be modest [15].

Current designs do not completely stall the pipeline on a fence while the write buffer drains. Instead, post-fence reads can speculatively load data. As long as no other processor observes the speculative read, no problem can occur. If an external processor does (i.e., it initiates a coherence transaction on the speculatively-read data), the local processor squashes the read and retries it. Unfortunately, even with speculation, not all the fence stall is removed: speculative reads cannot retire until after the write buffer is drained.

In this paper, we propose to target the elimination of all the stall. More aggressive reordering is acceptable as long as it does not result in an incorrect access order. Specifically, it is fine for a remote processor to observe local post-fence accesses before the local write buffer is drained, as long as *no dependence cycle* occurs — that is, as long as no SC violation occurs [23].

*This work is supported in part by NSF under grants CCF-1012759 and CNS-1116237, and by Intel under the Illinois-Intel Parallelism Center (I2PC).

†Abdullah Muzahid is now with the University of Texas at San Antonio.

ISCA'13 Tel Aviv, Israel

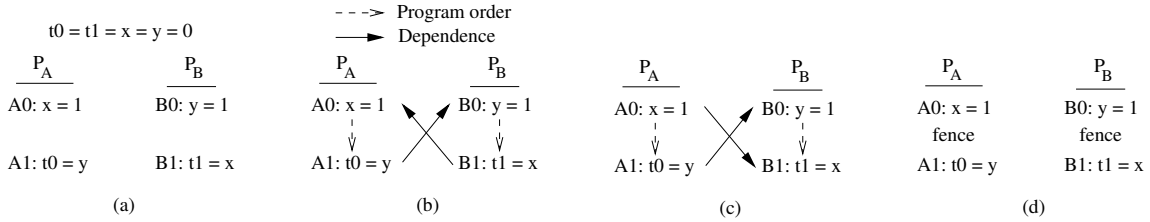

Figure 1: Pattern for SC violation.

In this paper, we present an aggressive fence design that we call *WeeFence*, or *WFence* for short. It allows post-fence accesses to skip the fence and proceed without stalling. Such accesses can typically complete and *retire* before the pre-fence writes drain from the write buffer. This is beyond today's most aggressive speculative fence implementations, where speculative reads cannot retire until after the write buffer is drained. Hence, WFence can save substantial time when write misses pile up before the fence. Only when an incorrect reordering of accesses is about to happen, does WFence stall until such a event cannot occur; in the large majority of cases, WFence induces zero visible stall. Finally, WFence is *compatible* with the use of conventional fences in the same program.

While the WFence idea can be used for any memory consistency model, we present the design for TSO [25]. This is because TSO is very similar to the model used by x86 processors [22]. We evaluate WFence using a simulated 8-processor multicore, and compare it to a conventional fence with speculation. We run parallel kernels that contain explicit fences and parallel applications that do not. For the kernels, WFence eliminates nearly all of the fence stall, reducing the kernels' execution time by an average of 11%. For the applications, a conservative compiler algorithm places fences in the code to guarantee that SC is not violated. In this case, on average, WFences reduce the resulting fence overhead from 38% of the applications' execution time to 2% (in a centralized WFence design), or from 36% to 5% (in a distributed WFence design).

This paper is organized as follows. Section 2 gives a background; Section 3 presents the WFence design; Sections 4 and 5 describe WFence's implementation; Sections 6 and 7 evaluate WFence; and Section 8 discusses related work.

2. BACKGROUND

2.1 Terminology, Consistency & Fences

Given an out-of-order processor, we are interested in a memory instruction's completion and retirement times. Following common terminology, we say that a read instruction completes (possibly speculatively) when the data loaded returns from the memory system and is deposited into a register. However, while the completed read remains speculative, it can still be squashed. A read retires when it is at the head of the Reorder Buffer (ROB) and has completed; after retirement, it cannot be squashed anymore. A write instruction retires when it is at the head of the ROB and its address and data are available. At this point, the write is effectively deposited (in order) into the write buffer to be merged with the memory system. The write completes (potentially out of order) when any coherence transaction that it triggers completes — e.g., when the local cache receives all the invalidation acknowledgements for the write. At this point, the write is removed from the write buffer.

The memory consistency model determines what orders of memory accesses are allowed. SC is the model that programmers have in mind when they program for shared memory. SC requires that the memory accesses of a multi-threaded program appear to be ordered in some global sequence as if the threads were multiplexed on a uniprocessor [11].

Current processors try to deliver high performance by using hardware mechanisms that reorder and overlap memory accesses. For example, processors that use Total Store Ordering (TSO) [25] allow the visible reordering of reads following writes. Specifically, while a write is waiting in the write buffer for completion (e.g., because of a cache miss), a subsequent read that reaches the ROB head can retire. This order relaxation can improve performance over a machine that enforces SC. TSO has been the choice in many commercial processors — e.g., x86 implements a model similar to TSO [22]. Consequently, this paper focuses on TSO.

In some cases, allowing such TSO reorderings results in a violation of SC. This is shown in Figure 1(a). Initially, variables *t0*, *t1*, *x*, and *y* are zero. Processor P_A first writes 1 to *x* and then reads *y* into *t0*, whereas processor P_B first writes 1 to *y* and then reads *x* into *t1*. Under SC, either *A1* or *B1* is the last access in the global order of accesses. Hence, after the execution of the code, either *t0* is 1, *t1* is 1, or both are 1. However, under TSO, it may happen that, while the *A0* write is waiting in the write buffer, the *A1* load reads the initial value of *y* and retires. Then, *B0* and *B1* execute. Finally, *A0* completes. We then have an effective order of *A1*, *B0*, *B1*, and *A0*. This causes both *t0* and *t1* to be zero, which is impossible under SC. It is an SC Violation (SCV).

Shasha and Snir [23] show what causes an SCV: overlapping data races where the dependences end up ordered in a *cycle*. Recall that a data race occurs when two threads access the same memory location without an intervening synchronization and at least one is writing. Figure 1(b) shows the order of the dependences at runtime that causes the cycle and, therefore, the SCV. In the figure, the source of the arrow is the access that occurs first. If at least one of the dependences occurs in the opposite direction (e.g., Figure 1(c)), no cycle (and therefore no SCV) occurs.

Given the pattern in Figure 1(a), Shasha and Snir [23] prevent the SCV by placing one fence between references *A0* and *A1*, and another between *B0* and *B1*. The result is Figure 1(d). Now, as we run the program in a TSO machine, *A1* waits for *A0* to complete, while *B1* waits for *B0*. As a result, either *A1* or *B1* is the last operation to complete, and there is no SCV.

2.2 Why Making Fences Very Cheap Matters?

Programmers and compilers insert fences in the code to prevent the compiler or the hardware from reordering memory accesses. The goal is to inexpensively manage the order in which the memory accesses of a thread are observed by other threads.

Programmers typically include explicit fences in performance-critical applications with fine-grained sharing. Examples include run-time systems, schedulers, tasks managers, and soft real-time systems. If fences were very cheap, programmers could improve the performance and scalability of these codes.

Compilers insert fences in codes to prevent incorrect reorderings. In particular, in high-level languages such as C++ or Java, the

programmer is allowed to employ intentional data races for performance, as long as the relevant variables are declared as *atomic* or *volatile*. Such declarations prompt the compiler to insert a fence after the access, which prevents any reordering by the compiler or hardware. Without the fences, some reorderings could be harmless, while others — like the one in Figure 1(b) — could cause SCVs.

If fences could be skipped while retaining correctness, programmers would not have to identify which accesses can cause data races. Instead, they could assume that all shared accesses can create races. They would declare all shared data as atomic, and the compiler would insert fences. The compiler would then automatically guarantee SC at little performance cost. The key is that hardware-induced reordering would not be curtailed. Moreover, while there would be a performance cost due to limiting compiler optimization, recent work has indicated that such effect may be modest [15].

2.3 Current Techniques to Speed-up Fences

Processors speed-up fences with in-window load speculation [8]. With this technique, a post-fence read can speculatively get the value from memory even while the fence is not completed — i.e., while pre-fence writes are not completed or pre-fence reads are not retired. The post-fence read cannot retire, but the processor uses its value while actively monitoring for any external coherence transaction on the cache line read. If such a transaction is received, the processor squashes and re-executes (at least) the post-fence read and its successors. The post-fence read can only retire after the fence completes. While the extent of speculation in real processors is unknown, our simulations use as baseline a processor with full in-window load speculation and exclusive store prefetch [8].

Conditional Fences [13] is a proposal to eliminate fence stall that, unlike our scheme, requires compiler support. The compiler analyzes the code and statically groups the fences into classes (called *Associate* fences) that prevent the same dependence cycle. At runtime, before a fence executes, the hardware checks if any of its associate fences is currently executing in another processor. Only if it is, the current fence stalls. Section 8 compares WFence to Conditional Fences and other proposals.

3. WFence DESIGN

3.1 Skipping Fences & Avoiding SC Violations

WeeFence, or *WFence* for short, is a new fence design that typically executes without inducing visible processor stall. It allows the memory instructions that follow the fence to proceed without stalling. While the WFence idea can be used for different memory consistency models, in this paper, we focus on a design for TSO [25] because TSO is very similar to the model used by x86 processors [22]. In this case, post-WFence read instructions can complete and retire before the WFence completes — i.e., before the pre-WFence writes complete.

This is beyond today's most aggressive speculative fence implementations, where post-fence read instructions can speculatively load data, but cannot retire until all of the pre-fence writes complete. As a result, a WFence can save substantial time when write misses pile up before the fence. In the large majority of cases, WFence induces no stall, as all of its actions are hidden by the ROB, and the instruction retiring rate is unaffected by the presence of a fence. Moreover, WFence is compatible with the use of conventional fences in the same program.

Since WFence enables aggressive memory access reordering, it needs to watch for incorrect reorders that lead to SC violations. For example, if any of the two fences in Figure 1(d) allowed its post-fence read to be ordered before its pre-fence write, an SC violation could occur. Hence, when WFence is about to allow a reorder that can lead to an SC violation, it stalls for a short period of time until such a condition cannot occur. This case, however, is rare.

WFence uses some extensions in the processor and memory system. They involve registering pending pre-fence accesses in a table, which other processors can check against post-fence accesses to see if there is a possibility for an SC violation. These actions reuse existing cache coherence protocol transactions.

3.2 Initial Design

Consider the basic pattern shown in Figure 1(d), where two fences are needed to avoid an SC violation. In Figure 2(a), we repeat the example and use WFences, therefore enabling reordering. To prevent an SC violation, WFence has to ensure that, if P_A:rd y happened before P_B:wr y (arrow (1)), then P_B:rd x stalls until it is ordered after P_A:wr x — and hence, arrow (2) is forced to point downward and no SC violation occurs.

A WFence involves two steps. First, the execution of a WFence instruction consists of collecting the addresses to be written by the pending pre-WFence writes, encoding them in a signature, and storing the signature in a table in the shared memory system called the *Global Reorder Table (GRT)*. We call such addresses the *Pending Set (PS)* of the WFence. The return message of such a transaction brings back from the GRT to the processor the combined addresses in the PSs of all the currently-active WFences in other processors — in a signature. The incoming signature is saved in a processor register called the *Remote Pending Set Register (RPSR)*.

In the second step, any post-WFence read compares its address against those in the RPSR. If there is no match, the read executes and may go on to eventually retire even before the WFence completes — a WFence completes when all pre-WFence accesses retire and complete, which requires that all pre-WFence writes drain from the write buffer. If, instead, there is a match, the read stalls. The stall lasts until all the remote WFences whose PSs are in the local RPSR complete. At that point, an arrow like (2) in Figure 2(a) cannot occur. When a WFence completes, it clears its GRT entry. Moreover, it needs to remove its PS addresses from any other processor's RPSR. This last requirement makes this initial design suboptimal; it is eliminated in Section 3.4.

The procedure described allows high concurrency by using conventional speculative execution. A post-WFence read instruction can execute even before the WFence has executed (and filled the local RPSR with the PS of all the other currently-active WFences). In this case, when the RPSR is finally filled, the read's address is compared to it, and the read is squashed if there is a match. The squashed read immediately restarts and, if it still matches, stalls. Moreover, when a post-WFence read stalls due to a match, subsequent local reads that do not match can still execute speculatively. However, because of the TSO model, they can only retire after the stalled read retires. Finally, a speculative read (stalled or otherwise) is squashed and restarted if it receives an external coherence access or if its line is evicted from the cache. Overall, the key insight is that a post-WFence read instruction can stop being speculative and retire *before* the earlier WFence completes; we will see when.

Figures 2(b) and (c) illustrate the algorithm. In Figure 2(b), WFence1 deposits its PS addresses in the GRT (1) and, since the GRT is empty, returns no addresses. P_A:rd y skips WFence1 (2) and executes because P_A's RPSR is empty. Later, in Figure 2(c), WFence2 deposits its PS addresses in the GRT (3) and returns the PS addresses of the active fences (4). At this point, an arrow like (1) in Figure 2(a) may have happened; hence WFence has to prevent an arrow like (2) in Figure 2(a). Therefore, as shown in Figure 2(c), as P_B:rd x tries to skip WFence2, it checks the local RPSR (5) and

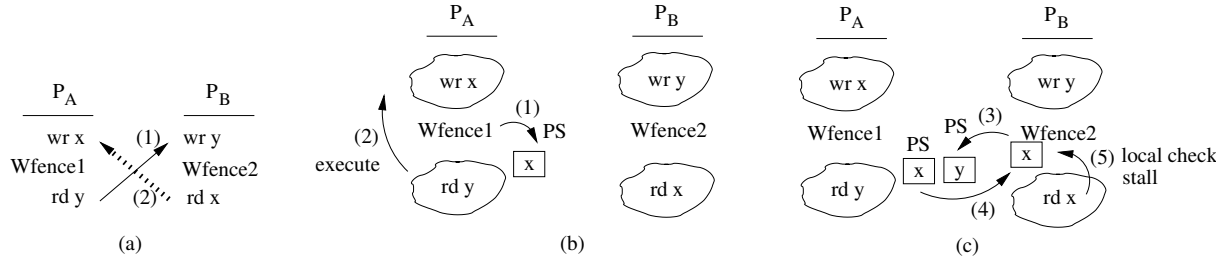

Figure 2: Averting an SC violation in a 2-WFence case.

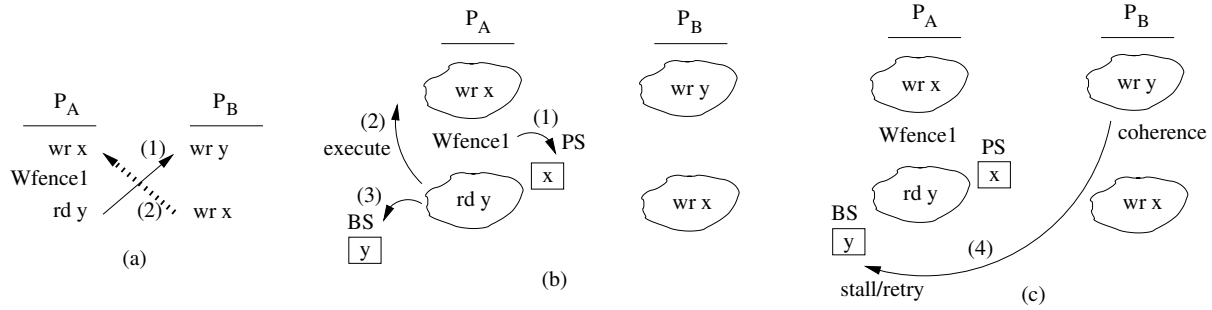

Figure 3: Averting an SC violation in a single-WFence case.

finds a match. It then stalls until WFence1 completes, removing the possibility of an arrow like (2) in Figure 2(a).

Stalling is rare, as it requires that two WFences dynamically overlap in time, that both threads access the same addresses in opposite sides of the fences, and that both dependence arrows threaten to go upward.

3.3 Complete Design

In the case discussed, both threads had fences to prevent the reordering of their accesses. However, a dependence cycle can occur even if only one of the threads reorders its accesses. Hence, in patterns where only one thread has a WFence, SC violations can still occur. In the case of TSO, the pattern is shown in Figure 3(a).

In Figure 3(a), assume that P_A:rd y happened before P_B:wr y (arrow (1)). TSO ensures that P_B:wr x is ordered after P_B:wr y. However, a reorder of the accesses in P_A (rd y ordered before wr x) could cause a dependence cycle. Hence, WFence has to ensure that, if P_A:rd y happened before P_B:wr y (arrow (1)), then P_B:wr x stalls until P_A:wr x has finished — preventing arrow (2).

This cycle cannot be avoided with the support described in Section 3.2. Since P_B has no fence, P_B:wr x does not know of (and cannot wait on) any remote PS. Instead, we must stall the consumer of the first arrow, namely P_B:wr y. For this, we will leverage the coherence transaction triggered by P_B:wr y, and stall the transaction until no cycle can occur.

To do so, we *extend* the WFence operation of Section 3.2 with two additional steps. We call the addresses read by the post-WFence read instructions executed before the WFence completes the *Bypass Set (BS)*. In the first step, as the post-WFence reads execute, they accumulate the BS addresses in a table in the local cache controller. Such table is called the *Bypass Set List (BSL)*.

Second, as external coherence transactions from other processors are received, their addresses are checked against the BSL. If there is a match and the local read is still speculative, conventional speculation automatically squashes the read and, therefore, we remove the read address from the BSL. However, if there is a match and the local read is *already retired* (although the WFence is not complete), the incoming transaction is not satisfied — it is either buffered or bounced. Later, when the local WFence completes, the BSL is cleared, and any transaction waiting on a BSL entry is satisfied. It is only at this point that the requesting access from the remote processor can complete. Any subsequent access in that processor can then proceed, but it is too late to create a cycle with the local processor because all pre-WFence writes have completed.

Figures 3(b) and (c) illustrate the algorithm for our example. In Figure 3(b), WFence1 deposits its PS addresses in the GRT (1). As P_A:rd y skips the fence and executes (2), it is part of the BS and hence saves its address (3). Later, in Figure 3(c), as P_B:wr y executes, it issues a coherence transaction to P_A. Assume that P_A:rd y has already retired and, hence, an arrow like (1) in Figure 3(a) will be generated. As the request arrives, it checks the BSL and matches. The request is either buffered or asked to retry. When WFence1 completes, the BS addresses are deallocated and the coherence request is satisfied. At this point, P_B:wr y completes. After this, as P_B:wr x completes, it cannot generate an arrow like (2) in Figure 3(a) because P_A:wr x has already completed.

3.4 Properties of the WFence Design

The WFence design resulting from the previous two sections has two key features. The first one is that when a WFence completes, it does *not* need to notify any other processor; the second is that conventional fences are seamlessly supported. We consider each in turn.

Recall from Section 3.2 that, when a WFence completed, in addition to clearing its GRT entry, it would have to clear its PS addresses in other processors' RPSRs. Now, thanks to the BSL, this is no longer required. Now, when a WFence completes, it only needs to clear its GRT entry. The other processors that contain the WFence's PS addresses in their RPSR will continue to wait (on a match) *only until their own* WFence completes. Once their own WFence completes, it can be shown that no dependence cycle is possible anymore and, therefore, they can clear their RPSR. This has the major advantage that no remote messages need to be sent, and the wait terminates on a local event.

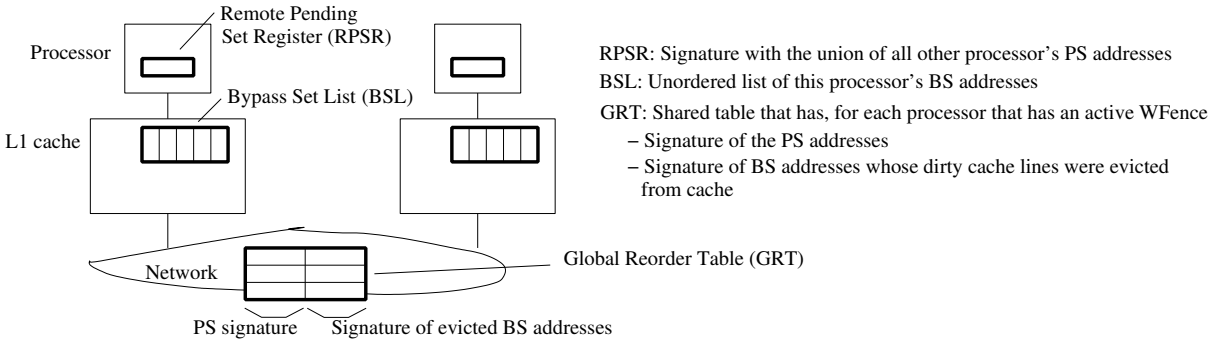

Figure 4: Multicore augmented with WFence hardware.

To see why, consider Figure 2(c) again. WFence2 is the second WFence to reach the GRT and it brings address *x* to its RPSR. P_B:rd x must wait on the RPSR, but only until WFence2 completes. At this time, P_B can safely clear its RPSR and let rd x commit, since no arrow like (2) in Figure 2(a) is possible. The reason is that: (i) if WFence2 is complete, then P_B:wr y completed; (ii) P_B:wr y completion required that the BSL of P_A had been cleared and, therefore, that WFence1 in P_A had completed; (iii) finally, if WFence1 had completed, then P_A:wr x must have completed and an arrow like (2) is not possible. In all cases, post-WFence reads waiting due to a match in the local RPSR to avoid a cycle can proceed as soon as their local WFence completes. A WFence never clears RPSR entries in other processors.

The second feature is that conventional fences are seamlessly compatible with the use of WFences in the same program. Indeed, a conventional fence affects a WFence as in the case described in Section 3.3: one of the interacting threads cannot reorder its accesses, either because the memory model prevents it (like in Section 3.3) or because there is a conventional fence.

3.5 Larger Numbers of Processors

The WFence algorithm is applicable to cycles with larger numbers of processors, irrespective of whether all processors use fences. In all cases, the cycle is averted. Figure 5 shows two examples with three processors: one with three WFences and one with just one. We now show that the cycles depicted cannot occur. This discussion requires that the WFences reach the GRT in a total global order. We later show how this is supported in a distributed GRT.

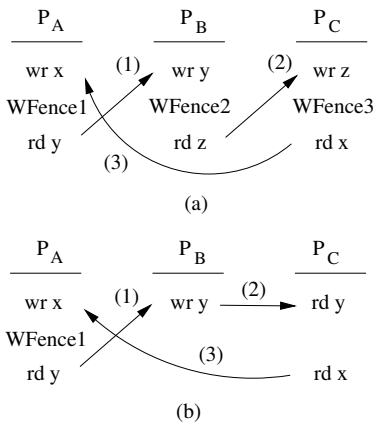

Figure 5: WFence use in patterns with several processors.

Figure 5(a) shows the case of three processors with three WFences. In the figure, we pick the WFence that reaches the GRT last. Let us assume that it is WFence3. There are two possible cases, depending on whether or not WFence3 returns address *x* from the GRT. First, if it does not, it means WFence1 has completed and, therefore, P_A:wr x has completed. Hence, arrow (3) is impossible and there is no cycle.

If, instead, the addresses returned by WFence3 include *x*, then P_C:rd x will wait until WFence3 completes (and clears P_C's RPSR). The completion of WFence3 implies that P_C:wr z completed which, according to dependence arrow (2), implies that P_B:rd z has retired and address *z* has been removed from P_B's BSL. Moreover, P_B's BSL is only cleared after WFence2 has completed. Following the same logic, WFence2's completion implies, following arrow (1), that WFence1 has completed and, therefore, that P_A:wr x has completed. Hence, arrow (3) is again impossible and there is no cycle.

Figure 5(b) has three processors with a single WFence. Consider P_C:rd x. TSO requires that it use the *x* value that exists after P_C:rd y retires — it can speculatively get an earlier value, but will be squashed and restarted if a writer such as P_A:wr x updates it in the meantime. The retirement of P_C:rd y implies that P_B:wr y completed (arrow (2)) which, in turn, implies that P_A:rd y has retired and address *y* has been removed from P_A's BSL (arrow (1)). This in turn implies that WFence1 has completed and, therefore, that P_A:wr x has completed. Hence, arrow (3) is impossible and there is no cycle.

Other cycles are averted in a similar way, including those with one or more conventional fences.

4. WFence IMPLEMENTATION

We now describe WFence's implementation in detail.

4.1 Hardware Structures

Figure 4 shows the three hardware structures needed to support WFences: RPSR, BSL, and GRT. The RPSR is a register in the processor that contains a signature generated with a Bloom filter. It is filled when the processor receives the response from the GRT to a WFence executed by the processor. The RPSR contains the union of the (line) addresses in the Pending Sets (PSs) of all the WFences that were active (i.e., were in the GRT) at the time when the processor executed the WFence (and visited the GRT). The RPSR is automatically cleared when the local WFence completes. There is also a functional unit that intersects the addresses of the post-WFence reads issued by the processor against the RPSR, and stalls the reads that match.

The BSL is a list of addresses in the cache controller. It stores the (line) addresses in the Bypass Set (BS), namely the addresses read by post-WFence read instructions that are executed by the processor while the WFence is incomplete. One such read instruction may be retired or still speculative. If the read is speculative, it will

be squashed (and removed from the BSL) in these four cases: (1) the response to the WFence execution fills the RPSR with a signature that includes the read's address, (2) the data loaded by the read receives a coherence transaction or (3) is evicted from the cache, or (4) the read is in a mis-speculated branch path. In all of these cases but the last one, the read is retried. The addresses of incoming coherence transactions are checked against the BSL. If one matches a BSL address for a retired read, then the coherence transaction is not allowed to complete — it is either stalled or bounced.

A dirty cache line that was accessed by a retired read in the BSL may be evicted from the cache. If we evicted it without taking any special action, the processor would not observe future coherence activity on the line. Consequently, when such a line is evicted, as it is written back to memory, its address is saved in the GRT entry of the processor. Since coherence transactions always check the GRT, the GRT will be able to stall (or bounce) future conflicting transactions on that address. To prevent overflow of these evicted entries in the GRT, they are encoded in the GRT in a per-processor signature. While these signatures may cause false-positive stalls, it can be shown that, if they use the same encoding as the RPSR, deadlocks are impossible.

Note that if the evicted cache line accessed by a retired read in the BSL was clean, no action is needed. Since the directory is not updated, the local processor will still observe future coherence activity on the line. Overall, in all cases, the BSL (and the processor's GRT entry) is cleared when the local WFence completes.

The Global Reorder Table (GRT) is a table in the memory system that is shared by all the processors. It is placed in a module that observes coherence transactions, such as the directory controllers in a distributed-directory system, or the bus controller in a snoopy-based system. It has at most one entry per processor. When a processor sends a WFence-execution message with its PS addresses to the GRT, the hardware creates an entry in the GRT. The entry contains the PS (line) addresses encoded in a signature. The entry is active until the WFence completes. The GRT entry for a processor may also contain a signature with (line) addresses from the WFence's BS. It contains the addresses (also present in the BSL) of dirty lines that were evicted from the cache due to a conflict.

Figure 6 shows how pre-WFence writes, WFence execution, post-WFence reads, and WFence completion interact with the hardware structures. For WFence execution and completion, we show the general case where a GRT access is needed. The next section describes the WFence operation in detail.

1. Pre−WFence write:
   Stall the request if it finds the matching address in a remote BSL
2. Local WFence execution (most general case):
   Put signature of own PS addresses in GRT
   Return from GRT a signature of the union of other processors' PS addresses and store it in the local RPSR
3. Post−WFence read:
   Stall the request if it finds a matching address in the local RPSR
   Else put address in the local BSL (if BSL is full, stall the read)
4. Local WFence completion (most general case):
   Clear the processor's entry in the GRT
   Clear the local RPSR and release any local reads waiting on it
   Clear the local BSL and release any external transactions waiting on it

Figure 6: Interaction with WFence hardware structures.

4.2 Timeline of a WFence Instruction

To clarify WFence operation, Figure 7(a) considers a code snippet with a write (W), a fence, and a read (R). It then shows a timeline of events using a conventional fence (b) or a WFence (c).

The three instructions enter the ROB in order. With a conventional fence (Figure 7(b)), the fence retires from the ROB (and completes) only when the write buffer is drained of W and of all the previous writes. If W or any of the previous writes miss, they directly add to the stall. After the fence retires, R retires.

With a WFence, there are three key times: WFence execution, WFence retirement, and WFence completion. They are shown in bold in Figure 7(c).

4.2.1 WFence Execution with/without GRT Access

As indicated in Figure 6, execution involves collecting the addresses to be written by the pending pre-WFence writes (the PS), encoding them in a signature, and sending the latter to the GRT. We can speed-up the code by executing the WFence as early as possible — i.e., as soon as all the PS addresses are known. However, in doing so, we will induce a network access to the GRT. In reality, if none of the pre-WFence writes ends up missing in the cache, there will be no need to access the GRT.

Consequently, we do not immediately execute a ROB-resident WFence. Instead, only when the write buffer flags a new cache miss, does the hardware check for any subsequent WFence that has not yet executed. It is only then that the WFence starts collecting the PS addresses and will issue a network access. As we will see, if an unexecuted WFence reaches the ROB head and finds that the write buffer is empty, it does not need to access the GRT.

Figure 7(c) shows the case when the WFence executes early (due to a write miss). In the meantime, R executes and, before the processor receives the response from the GRT, R completes. In this case, R remains speculative. As soon as the GRT response arrives and updates the RPSR, R compares its address to it. Typically, the RPSR is null or contains very few addresses.

4.2.2 WFence Retirement

Eventually, W reaches the ROB head and retires into the write buffer. After this, the WFence reaches the ROB head. If the WFence has already executed, it also retires, as shown in Figure 7(c). Retirement involves removing the WFence from the ROB and placing it in the write buffer after W. Immediately after, R reaches the ROB head and retires. Comparing R retirement points in Figures 7(b) and (c), we see that WFence can save substantial time.

Two special cases can occur when the WFence reaches the ROB head. One is that the response from the GRT has not yet been received and the write buffer is not empty. In this case, WFence stalls until it receives the response (if the write buffer is empty, WFence retires immediately).

The second case is if the WFence is still unexecuted. If so, WFence checks the write buffer. If the write buffer is empty or contains a single cache hit, then WFence retires without needing a GRT access. This is a common case. Otherwise, the WFence executes as usual: it collects all the addresses in the write buffer and sends them as a signature to the GRT. This situation is the most unfavorable one, since it delays the retirement for the latency of a round trip to the GRT. Still, this latency is typically substantially lower than the one added by the cache miss of W in conventional fences: the write miss may involve a costly coherence transaction and even a main memory access.

4.2.3 WFence Completion

After W completes, the WFence reaches the write buffer head and completes (Figure 7(c)). WFence completion involves deleting the WFence entry from the write buffer, clearing the local BSL and, if a GRT entry had been allocated, clearing the entry and also the RPSR.

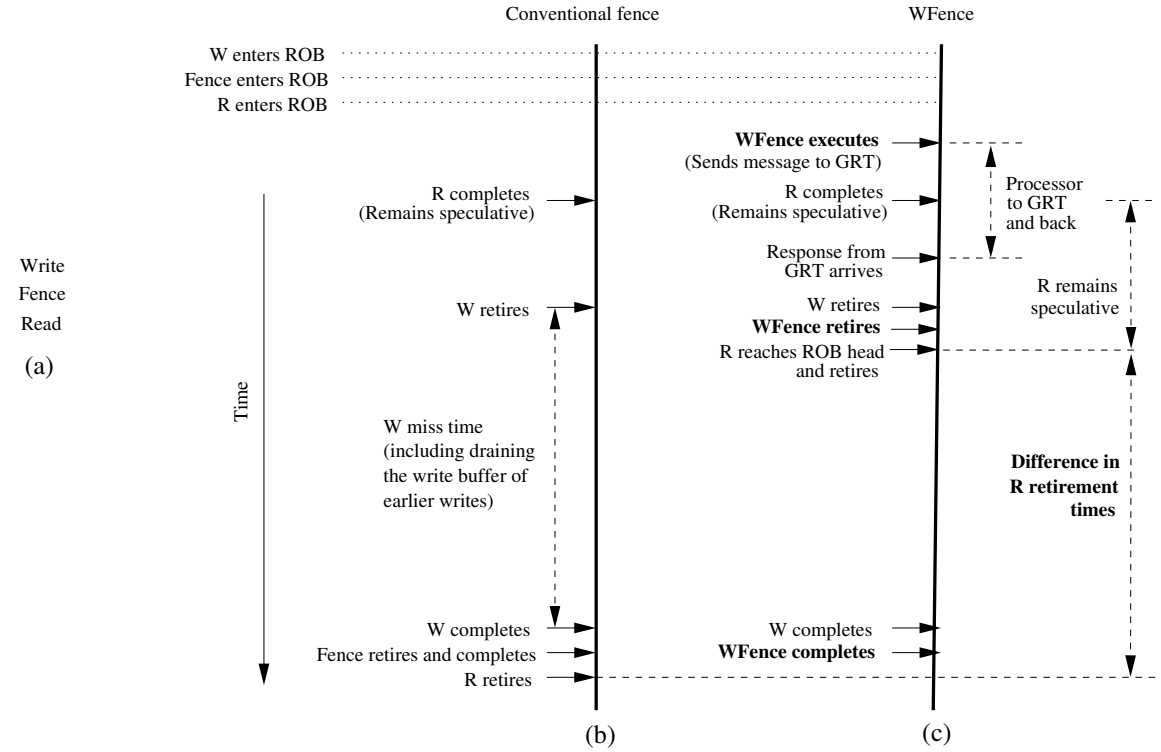

Figure 7: Timeline of a WFence instruction.

Overall, in the common case of early WFence execution or no pre-WFence write miss, WFence does not stall the retirement rate at all. The worst case occurs if the WFence remains unexecuted, reaches the ROB head, and needs a round trip to the GRT. Still, post-WFence reads can continue executing speculatively.

5. ADDITIONAL ISSUES

5.1 No Deadlock is Possible

With WFences, there is no possibility of deadlock. To see why, consider first two processors. Based on WFence's operation from Section 3, one can think of the two cyclic stalls shown in Figure 8: when the two reads wait on the two writes (a), or when the two writes wait on the two reads (b).

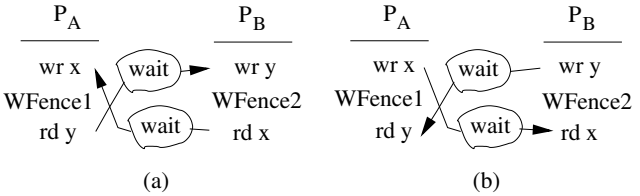

Figure 8: Potential cyclic stalls with two processors.

The case in Figure 8(a) can occur, but it does not deadlock. The reason is that the writes eventually complete, then the WFences complete, and then the RPSRs are cleared, releasing the reads.

The case in Figure 8(b) cannot occur. To see why, assume that it occurs, and pick the last fence to reach the GRT (say WFence2). Address x from P_A:wr x was put by WFence1 in the GRT, and was fetched by WFence2. As a result, P_B:rd x sees a match with its local RPSR and, rather than inserting its address in P_B's BSL, stalls until WFence2 completes — which requires that P_A:wr x has completed. Hence, P_A:wr x will not be stalled by P_B:rd x because P_B:rd x will not be in P_B's BSL. Even if P_B:rd x managed to put itself in the BSL speculatively, it would be squashed, either when P_A:wr x executed or when WFence2 filled the local RPSR with address x. Hence, this case cannot occur.

Deadlocks with more than two processors can be shown to be impossible in the same way.

5.2 Value Forwarding

Processors perform value forwarding, whereby a read obtains data from an earlier write in the pipeline that has not yet completed. In this case, the corresponding memory line may not be in the cache yet. Recall from Figure 3(c) that WFence requires that a speculative post-WFence read use its address to stall incoming coherence transactions. This is the role of the BSL. However, if such a read gets its data from an incomplete pre-WFence write, such a transaction may not occur. Consequently, we require that, before the post-WFence read that wants the data forwarded can execute, the pre-WFence write bring the line into the cache.

5.3 Distributed Global Reorder Table (GRT)

For a small multicore, we use a centralized GRT associated with the bus controller. For a larger machine, we propose a scalable design of a distributed GRT. The GRT is distributed like a directory, broken down into modules in charge of address ranges. Each module is associated with the corresponding directory module.

With such a design, as we follow the algorithms for WFence execution and completion in Figure 6, we see that a WFence may need to communicate with multiple GRT modules. Specifically, it needs to deposit a signature in all of the modules that map any address in its Pending Set (PS); it needs to read a signature from all of the modules that map any address that may be in its Bypass Set (BS). Unfortunately, such a distributed protocol is prone to races

when multiple processors concurrently communicate with sets of GRT modules. Hence, we radically simplify the algorithm.

Our simplifications rely on several observations. First, data accesses tend to have spatial locality. In addition, we assign address ranges to GRT modules at page-level granularity, and use a first-touch page allocation policy. As a result, a WFence's PS often maps to a single GRT module. Similarly, the WFence's BS will often map to the same GRT module as its PS. Furthermore, if there is no need for the WFence to perform a GRT access (a common case), then it does not matter where the BS maps to. Finally, it is always correct for a WFence to operate as a conventional fence.

With these insights, we design the WFence execution algorithm for the distributed GRT as follows. As the WFence collects its PS addresses, it determines whether they all map to a single GRT module. If they do, then the WFence executes as usual, communicating with the single GRT module. In this case, which is the common one, there are no race concerns.

Otherwise, the WFence works as a conventional fence: the GRT is unused and post-fence reads remain speculative until the fence completes. This approach ensures that there are no multiple WFences racing to create multiple GRT entries with inconsistent state.

In all cases, post-WFence reads execute speculatively as usual, without any concern about the GRT distribution. However, when a read is at the ROB head ready to retire after a fence that executed as a WFence, it checks two conditions: (i) whether the WFence communicated with any GRT module and, (ii) if so, whether the read maps to the same GRT module. The very common case is that either it did not communicate with any module or, if it did, the module is the one where the read maps to, and the read address is not in the RPSR. In this case, the read retires immediately as in our centralized WFence. Otherwise, the read will not retire until the WFence completes — preventing the retirement of subsequent accesses.

5.4 Multiple WFences per Thread

When a processor is processing a WFence and allowing subsequent memory operations to execute, it may find a second WFence in the instruction stream. A naive approach is to stall the pipeline until the first WFence completes. However, this approach has overhead. Consequently, we use a design that allows multiple active WFences in a processor.

In this design, a processor dynamically assigns a tag to each WFence it executes (e.g., two bits if we allow up to four active WFences). As the hardware deposits a signature in the GRT and brings back a signature for the RPSR, it tags the GRT entry and the RPSR with the WFence tag. Each address in the BSL is also tagged with the latest WFence tag in program order. When a second WFence executes, the GRT entry and RPSR are overwritten, in both data and tag, since they now encode more up-to-date state. The BSL entries are retained, but when reads after the second fence add addresses to the BSL, they use the new tag. Later, when a WFence completes (recall that WFences complete in program order), it clears the GRT entry and RPSR only if they have its own tag. Moreover, it only clears the BSL entries that have its own tag.

Figure 9(a) shows an example with two WFences. Figure 9(b) shows the state of the data structures after WFence2 has executed but before WFence1 has completed.

5.5 Filtering Private Accesses

Recently, to reduce the cost of enforcing SC, researchers have proposed constraining memory reordering only for shared data [24]. We use this approach for WFence. Specifically, if a reference accesses an address that is clearly private to the thread, then the reference is not included in the Pending or Bypass Sets of a WFence; such an address cannot be involved in any inter-thread conflict and, therefore, in any SC violation. Hence, we do not store such an address in the GRT, RPSR, or BSL. As a result, these structures are more precise and less information is transferred.

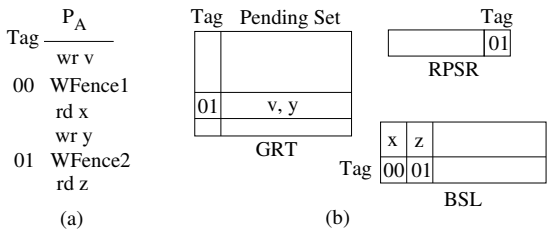

Figure 9: Example of two active WFences in a processor.

We conservatively infer the private properties of a variable only statically. A more aggressive approach is used in [24], where the properties of a variable are dynamically tracked at page level with hardware extensions in the TLB and OS support.

5.6 Application to Release Consistency

The WFence idea can also be applied to systems that support other memory consistency models, although it may need some modifications. As an example, in this section, we outline at a high level the design for a system supporting release consistency (RC).

Under RC, the hardware can reorder rd-rd and wr-wr accesses, in addition to wr-rd as in TSO. Hence, the program may have WFences between such pairs of accesses. While, in theory, RC also allows rd-wr reorder, current processor hardware does not deposit the write into the write buffer until the prior read has retired. As a result, programs will not have an explicit fence between a read and a subsequent write even if the rd-wr order is required for correct operation. This means that, like under TSO, WFence under RC also has to be designed to avert SC violations in code patterns where only one thread has a WFence. This means that the BSL concept is also needed under RC.

The Pending Set of a WFence is the addresses of both the non-completed writes and the non-retired reads preceding the WFence when the WFence executes. The Bypass Set is the addresses of the following post-WFence accesses: the reads executed and the writes deposited in the write buffer before the WFence completes.

6. EVALUATION SETUP

For our evaluation, we perform detailed cycle-level execution-driven simulations using SESC [21]. We model a multicore with 8 processors connected in a mesh network with a directory-based MESI coherence protocol under TSO. Each core has a private L1 cache, a bank of a shared L2 cache, a portion of the directory and the corresponding module of the distributed GRT. For a few experiments, we connect the 8 processors in a bus and use a MESI coherence protocol with a centralized GRT. In all cases, we use private data access filtering for the WFences. For the addresses in the GRT, RPSR, and BSL, we use cache line addresses. Table 1 shows the architecture parameters. Based on these parameters, the storage overhead added by WFence is 64B for each core's RPSR, 128B for each L1 controller's BSL, and 1KB for the GRT.

We compare two multicore architectures: one with WFences (*WFence*) and one with conventional fences that support in-window load speculation and exclusive store prefetching [8] (*Baseline*). We use a distributed GRT as the default, and run a few experiments with a centralized GRT. We model traffic and resource contention in the system. We run two sets of programs. The first one is 6

Architecture	8-core multicore
Core	Out of order, 3-issue wide, 2.0 GHz
ROB & wr-buffer	104-entry ROB & 64-entry FIFO write-buffer
L1 cache	Private 32KB WB, 4-way, 2-cycle RT, 32B lines
L2 cache	Shared 1MB WB with eight 128KB banks A bank: 8-way, 11-cycle RT (local), 32B lines
Cache coherence	MESI under TSO (if directory: full mapped)
On-chip network	3x3 2D-mesh: 5 cycles/hop, 256bit links or bus: 25 cycles/transaction (avg), 256bit bus
Off-chip memory	Connected to one network port, 200-cycle RT
Address mapping	Page-level granularity, first-touch policy
WFence	1 cycle to encode an address into signature, 4 active WFences per processor
GRT module	Together with the L2 tags (if distributed) or with bus controller (if centralized)
GRT/RPSR signat.	512 bits, in 4 128-bit Bloom filters with H3
BSL	Up to 32 entries per processor, 4B per entry

Table 1: Multicore architecture modeled. RT stands for round trip from processor.

programs that we obtain from [3, 4, 7] and use explicit fences for correctness (Table 2). We call these programs *kernels*. We study the performance that *WFence* attains over *Baseline*.

bakery	Mutual exclusion algorithm for arbitrary # of threads
dekker	Mutual exclusion algorithm for two threads
lazylist	Concurrency list algorithm using bakery lock
ms2	Concurrent queue algorithm using bakery lock
peterson	Mutual exclusion algorithm for arbitrary # of threads
workstead	Cilk THE work stealing algorithm

Table 2: Kernels evaluated that use explicit fences.

The second set is 17 C/C++ programs from SPLASH-2 and PARSEC, and pbzip2 (parallel bzip2). While these applications run correctly under TSO, we use them to emulate (very conservatively) what would be done to guarantee SC in arbitrary programs if fences were very cheap. Specifically, we use LLVM to simply turn every access to potentially shared data into an *atomic* access. This prevents the compiler from reordering across these accesses, inducing a measured execution overhead of about 4% on average (which is consistent with [15]). In addition, as the compiler generates the binary code, it inserts a fence after each atomic write, to prevent the hardware from reordering it with any subsequent read. The compiler uses a simple algorithm to remove fences between two consecutive writes — they are unnecessary, since TSO would not reorder across them. Overall, these transformations guarantee SC for the programs. We call the transformed code *SC-apps*. In our evaluation, we study the performance overhead of these fences using either *WFence* or *Baseline*.

It is interesting to compare our simulated *Baseline* fence to the actual fence in an Intel machine. For this, we write small programs using the *mfence* instruction [10] (which we find has a cost similar to *xchg*). We run tests on a desktop with an 8-threaded Intel Xeon E5530 processor. Figure 10 shows the visible stall latencies induced by the native Intel fence and the simulated fence in *Baseline*.

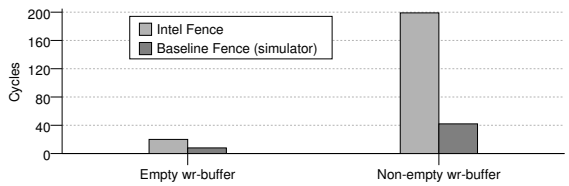

Figure 10: Comparing a native Intel fence to a simulated one.

With an empty write buffer, the Intel fence costs 20 cycles, while our fence in *Baseline* costs 8. With a write buffer filled with many entries, the Intel fence costs 199 cycles, while our fence in *Baseline* costs 42. Since there are many factors that affect a fence's overhead, and the microarchitecture of the Intel fence is unknown, we do not seek to tune our simulator to model the Intel fence.

7. EVALUATION

7.1 Performance with Centralized GRT

Figure 11 compares the execution time of the kernels on the *Baseline* and *WFence* multicores with the centralized GRT. Kernel execution times are normalized to those in *Baseline*, and broken down into time stalled in fences (*Fence*) and the rest (*Useful*). The figure shows that, in *Baseline*, these kernels spend an average of 12% of their time stalled in fences. *WFence* eliminates most of such stall, reducing the execution time of the kernels by an average of 11%. This shows the effectiveness of our new fence.

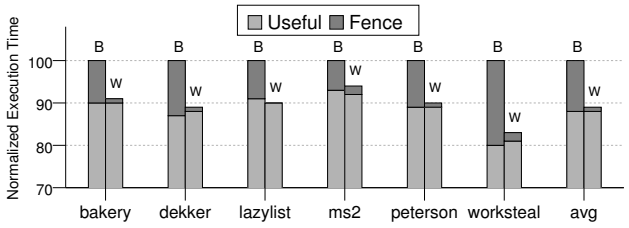

Figure 11: Performance impact on kernels for centralized GRT. In the figure, B and W refer to the *Baseline* and *WFence* multicores.

Figure 12 shows the execution time overhead induced in the applications by transforming them into SC-apps, conservatively guaranteeing SC. The overheads come from limiting compiler-induced optimization (*Compiler*) and reducing hardware-induced reordering (*Fence*). *Compiler* is largely the same in both *Baseline* and *WFence* multicores, and adds, on average, 4% overhead. For some codes, limiting compiler optimization slightly improves the speed (*Compiler* is negative), causing the bar to start lower than zero.

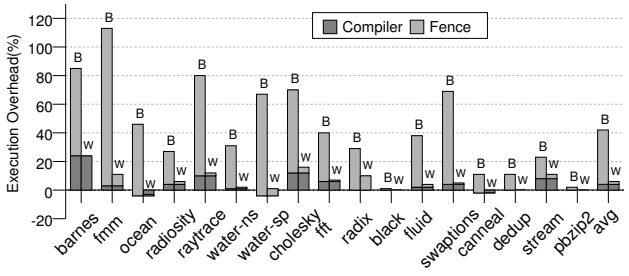

Figure 12: Execution overhead of conservatively guaranteeing SC for centralized GRT. B and W mean *Baseline* and *WFence* chips.

The *Fence* overhead is much larger in *Baseline* than in *WFence*. On average, limiting hardware reordering for guaranteed SC adds a fence overhead of 38% in *Baseline* and of only 2% in *WFence*. Overall, we estimate that, with WFences, the cost of conservatively guaranteeing SC in programs on TSO hardware is very small.

7.2 Performance with Distributed GRT

Figures 13 and 14 repeat Figures 11 and 12 for multicores with the distributed GRT. We see that the trends are the same and the values are very close. In Figure 13, the kernels spend an average of 11% of their time in fences, and *WFence* reduces the execution time by an average of 9%.

Codes	#Fences/1K inst		Addr in RPSR		Addr in BSL		#wr stalled by BSL		#rd stalled by RPSR		#BSL disp. per fence	Number of GRT Modules				Traffic inc. (%)
	Stat.	Dyn.	Avg	Max	Avg	Max	Avg	Max	Avg	Max		PS		Add'l BS		
												Avg	Max	Avg	Max	
barnes	78	46	1.1	17	2.0	8	0.0	2	0.0	3	0.0	1.04	6	0.0	2	11
fmm	51	54	1.6	23	1.2	27	0.0	3	0.0	2	0.0	1.07	4	0.3	3	14
ocean	46	8	5.3	58	1.4	32	0.0	1	0.0	4	0.0	1.31	4	0.0	1	9
radiosity	34	16	2.3	22	1.4	32	0.0	5	0.1	2	0.0	1.02	6	0.1	1	13
raytrace	75	42	1.4	14	2.0	9	0.0	1	0.0	2	0.0	1.03	5	0.0	2	11
water-ns	55	7	2.0	10	1.0	18	0.0	5	0.1	1	0.0	1.00	3	0.0	2	5
water-sp	48	39	3.1	17	2.1	24	0.0	4	0.0	1	0.0	1.01	4	0.0	2	1
cholesky	37	33	1.2	18	1.1	14	0.0	4	0.1	2	0.0	1.03	4	0.0	5	9
fft	25	51	3.6	23	2.1	12	0.0	1	0.0	2	0.0	1.17	3	0.0	1	7
radix	26	11	6.5	36	0.4	12	0.0	2	0.0	1	0.0	2.42	6	0.0	4	19
black	24	2	0.3	2	0.4	1	0.0	0	0.0	1	0.0	1.00	2	0.4	2	10
fluid	28	18	1.7	9	1.0	3	0.0	1	0.0	2	0.0	1.12	5	0.2	4	11
swaptions	35	26	3.5	14	0.1	26	0.0	3	0.0	4	0.0	1.86	6	0.0	6	22
canneal	24	12	1.4	6	1.4	16	0.0	1	0.1	2	0.0	1.08	5	0.1	2	0
dedup	42	13	1.2	8	0.2	13	0.0	0	0.0	1	0.0	1.01	3	0.0	2	9
stream	29	2	0.8	5	3.0	24	0.0	4	0.1	2	0.0	1.00	3	1.5	3	15
pbzip2	30	27	0.1	3	0.7	7	0.0	2	0.0	3	0.0	1.00	3	0.0	2	5
Average	40.4	23.9	2.2	16.8	1.3	16.4	0.0	2.3	0.0	2.1	0.0	1.18	4.2	0.1	2.6	9.7
bakery	3	2	1.0	3	0.9	2	0.0	2	0.1	2	0.0	1.00	2	0.0	3	4
dekker	23	3	0.1	2	1.0	4	0.0	1	0.0	1	0.0	1.01	1	0.0	2	8
lazylist	1	4	0.8	4	0.8	3	0.0	2	0.0	1	0.0	1.00	3	0.1	2	6
ms2	3	6	0.2	2	0.7	2	0.0	2	0.1	2	0.0	1.00	2	0.0	1	5
peterson	3	3	0.4	2	1.1	5	0.0	1	0.0	2	0.0	1.00	2	0.0	1	5
worksteal	8	7	1.1	3	1.0	9	0.0	3	0.0	4	0.0	1.08	4	0.1	4	9
Average	6.8	4.2	0.6	2.7	0.9	4.2	0.0	1.8	0.0	2.0	0.0	1.02	2.3	0.0	2.2	6.2

Table 3: Characterization of WFence. The addresses used are (32-byte) *line* addresses.

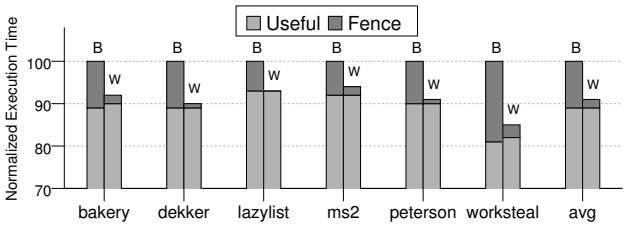

Figure 13: Performance impact on kernels for distributed GRT. In the figure, B and W refer to the *Baseline* and *WFence* multicores.

In Figure 14, the *Compiler* overhead is largely the same as for the centralized GRT. Its average value is 4%. The *Fence* overhead also follows the same trend. On average, WFence reduces *Fence* from 36% in *Baseline* to only 4% in *WFence*. The overhead in *WFence* largely comes from three codes: radix, swaptions, and ocean. For these codes, the Pending Set of WFences sometimes maps to multiple GRT modules, thus transforming a WFence into a conventional fence.

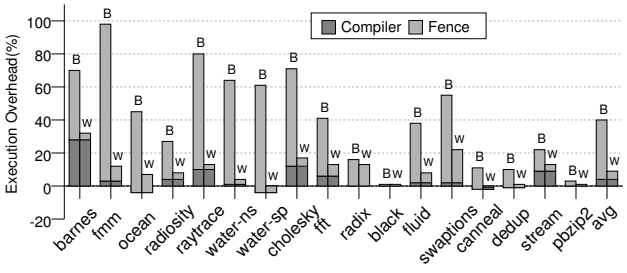

Figure 14: Execution overhead of conservatively guaranteeing SC for distributed GRT. B and W mean *Baseline* and *WFence* chips.

Figure 15 shows the execution overhead of running SC-apps natively on the 8-threaded Intel-based desktop. Conservatively guaranteeing SC adds, on average, 4% overhead due to compiler restrictions and 58% due to hardware restrictions.

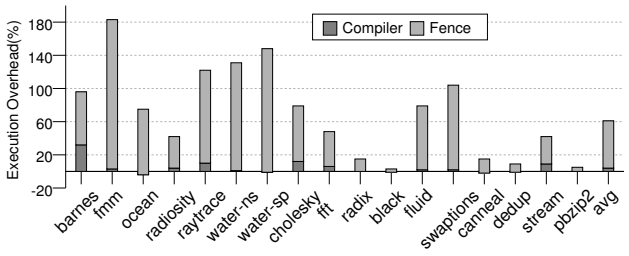

Figure 15: Execution overhead of conservatively guaranteeing SC in an Intel-based desktop.

7.3 Characterizing WFences

In the rest of the paper, we focus on the multicore with distributed GRT. Table 3 characterizes our proposed fence in the *WFence* multicore with 8 processors. We proceed from left to right columns.

Columns 2-3 show the number of fences inserted or present per 1K instructions, statically and dynamically. For SC-apps, the numbers are very high (average 24 dynamically). The reason is that our compiler pass is very naive. A more precise analysis can reduce the number of fences and further decrease the cost of guaranteeing SC in Figure 14. In kernels, fences are few and well tuned (average 4 dynamically).

Columns 4-5 show the number of line addresses that are encoded in the 512-bit signature brought-in from the GRT by a WFence and stored in the local RPSR. We show average and maximum values. On average, few addresses are encoded (2.2 in SC-apps and 0.6 in kernels). The maximum value can be a few tens. As a result,

very few reads stall due to false-positive collisions with the RPSR. Specifically, it can be shown that, on average, only 4.2% of the stalled reads in SC-apps and 2.5% in kernels are stalled due to false positives. As a related fact, when a WFence in SC-apps sends its Pending Set to the GRT, the average number of line addresses that it includes is only about 1. The reason is because of data locality and the fact that we use line addresses.

Columns 6-7 show the average and maximum number of line addresses in the BSL list of a WFence. On average, the number is small (1.3 for SC-apps and 0.9 for kernels). The reason is again because of data locality and the use of line addresses. Increasing the ROB depth and issue width can directly increase these numbers. The maximum number is 32 because this is the size of our BSL.

Columns 8-9 show the average and maximum number of remote writes stalled by the local BSL for each WFence. The average number is 0.0. Since very few write stalls are observed, they do not introduce any visible performance overhead.

Columns 10-11 show the average and maximum number of local post-WFence reads that are stalled or squashed by the RPSR fetched by a WFence. The average number is 0.0. This confirms our hypothesis that conflicts across concurrent fences are rare.

Column 12 shows the number of displaced dirty cache lines whose address is in the BSL. This is a very rare event. Hence, very few BSL entries are moved to the GRT.

Columns 13-16 examine the locality of accesses to the modules of the distributed GRT. First, we consider the addresses in a WFence's Pending Set (PS). For these addresses, Columns 13-14 show the average and maximum number of GRT modules where they map. On average, PSs in the kernels map to 1.02 modules; in the SC-apps, they map to 1.18 modules. The highest numbers are in radix, swaptions, and ocean, where the average is 2.42, 1.86, and 1.31, respectively. Because of this relatively bad spatial locality, WFence works less well in these applications. This is confirmed by the higher *Fence* overhead in Figure 14.

Second, we consider a WFence's Bypass Set (BS). For these reads, Columns 15-16 show the average and maximum number of additional GRT modules where they map. These are the modules beyond the one(s) where the PS maps. On average, these modules are few: 0.0 in the kernels and 0.1 in the SC-apps. The one exception is streamcluster, where the average is 1.5.

The last column is the increase in bytes transferred in the network due to WFences— mostly due to communication with the GRT on WFence execution and completion. The average increase is 10% for SC-apps and 6% for kernels. This traffic volume is modest and causes no significant congestion. Moreover, it is distributed toward different GRT modules.

7.4 Scalability with the Number of Processors

Figure 16 shows the execution time overhead induced in the applications by transforming them into SC-apps for different processor counts. The figure shows runs for 2, 4, 8, and 16 processors. As in Figure 14, the overhead includes compiler- and hardware-induced effects. Charts (a) and (b) correspond to the *Baseline* and *WFence* multicores. Due to lack of space, the charts show bars for only some codes, but the averages are for all 17 SC-apps.

The data shows that, as the processor count increases, the average overhead tends to increase only very slowly. This applies to both the *Baseline* and *WFence* multicores. Overall, we see that *WFence* is scalable.

8. RELATED WORK

As described in Section 2.3, Conditional Fences (C-Fence) [13] is a scheme to reduce fence overhead dynamically. It has four major

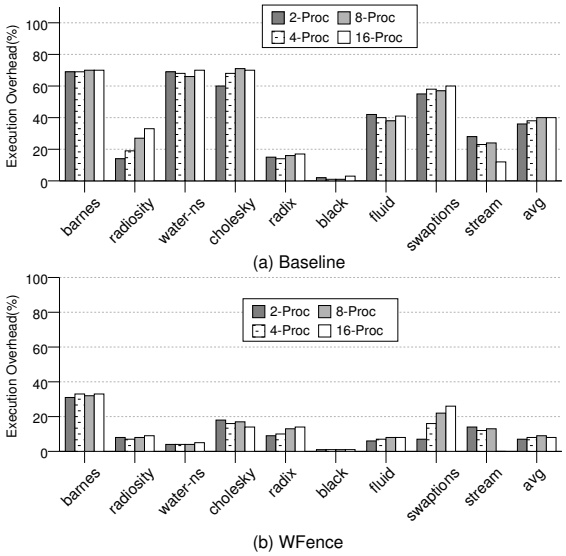

Figure 16: Scalability of conservatively guaranteeing SC.

differences with WFence. First, C-Fence requires a compiler pass to determine Associate fences, whereas WFence does not use any compiler. Second, Associate fences are grouped conservatively: even if two fences do not separate the same set of addresses, they may be put in the same group (e.g., when they were placed in the code to break a more-than-2-variable dependence cycle). As a result, a fence may stall for an Associate fence even if there is no potential SC Violation (SCV). WFence overcomes this limitation by dynamically checking for address conflicts. Third, WFence works seamlessly with conventional fences, while C-Fence does not. This is because C-Fence needs the information about fence association. Finally, the C-Fence compiler pass sometimes needs to insert fences where none was needed (e.g., in Figure 3(a)), to be able to create Associates, while WFence never adds new fences.

Another approach to reduce stalls due to the memory consistency constraints is post-retirement speculation (e.g., [1, 5, 9, 20, 27]). This technique retires accesses speculatively, buffering their state. Often, the speculative accesses are committed as a group or rolled back together. This approach requires a larger storage for speculative state, often using the L1 for it. It needs checkpointing and rollback of large state. Moreover, it often needs modifications to the coherence protocol and cache structures. Finally, it keeps post-fence reads speculative for a longer period, risking more squashes due to external coherence requests or local cache displacements. WFence shortens the speculative window, reducing squashes.

Four recent related works with different goals are Conflict Ordering (CO) [14], End-to-End SC [15, 24], Vulcan [17], and Volition [18]. CO's goal is to ensure SC execution on a relaxed-consistency platform. It allows accesses to bypass prior pending accesses if there is no potential for SCV; otherwise, it stalls them. CO has three main differences with our work. First, CO assumes that the program may have SCVs and tries to ensure SC while achieving high performance; we assume that the program has the necessary fences for SC and try to reduce the overhead of these fences. Second, CO requires every cache miss to bring pending write information from the directory, whereas we only bring the PSs when a WFence executes. Third, while CO works well for RC, it is likely suboptimal for TSO: to retire a read, CO needs to know whether any of its preceding writes missed and, if it did, it needs its pending write information. However, in TSO, writes are serialized, which serializes this information. WFence has no such requirement.

End-to-End SC's goal is to ensure SC from the source level. Its SC-preserving compiler [15] and its SC hardware [24] prohibit any

reordering of shared accesses, but allow private accesses to be reordered. WFence is different in that: (i) it focuses on pre/post-fence accesses only, and (ii) for these accesses, it is more aggressive than End-to-End SC, since it allows *shared* accesses to be reordered without causing SCVs.

The goal of Vulcan [17] and Volition [18] is to detect SCVs in executions on relaxed consistency platforms. They try to find a dependence cycle in hardware and trigger an exception when the cycle is found. They use a different approach than WFence. They create graphs of dependences to find cycles between processors. Vulcan is designed for centralized systems and Volition for scalable systems and cycles with any number of processors.

WFence is also related to proposals to eliminate or reduce the cost of synchronization operations, such as Speculative Lock Elision [19] or Speculative Synchronization [16]. These proposals differ from WFence in that they do not focus on optimizing an individual fence, but a whole critical section or barrier operation.

Software researchers have built on the cycle-detection algorithm of Shasha and Snir [23] to insert fences in codes running on relaxed consistency platforms and guarantee SC. Their goal is to minimize the number of fences introduced to guarantee SC. They rely on extensive compiler analysis (e.g., [12, 26]) or on off-line runs of data-race detectors [6]. While the slowdowns resulting from guaranteeing SC are sometimes significant (e.g., *Baseline* in our Figure 14), researchers have been able to progressively reduce them. WFence is a *complementary* approach to help them minimize the overhead of SC guarantees.

9. CONCLUSIONS

Today's fences can be quite expensive. If, instead, they were largely free, software could benefit substantially: programmers could write faster fine-grained concurrent algorithms, and C++ and Java compilers could guarantee SC at little cost.

In this paper, we have presented WFence, a fence that is very cheap because it allows post-fence accesses to skip it. Such accesses can typically complete and retire before the pre-fence writes have drained from the write buffer. If an incorrect access reordering is about to happen, the hardware stalls for a short period to avoid it. In addition, WFence is compatible with the use of conventional fences in the same program.

We presented the WFence design for TSO, and compared it to a conventional fence with speculation for 8-processor simulations. We ran parallel kernels that contain explicit fences and parallel applications that do not. For the kernels, WFence eliminated nearly all of the fence stall, reducing the kernels' execution time by an average of 11%. For the applications, a conservative compiler algorithm placed fences in the code to guarantee SC. Then, on average, WFences reduced the resulting fence overhead from 38% of the applications' execution time to 2% (in a centralized WFence design), or from 36% to 5% (in a distributed WFence design).

Overall, the resulting cheap fence can be a good help for parallel programming. In our future work, we plan to optimize the distributed GRT design for the case where a WFence maps to multiple GRT modules.

10. REFERENCES

[1] C. Blundell, M. M. K. Martin, and T. F. Wenisch. InvisiFence: Performance-Transparent Memory Ordering in Conventional Multiprocessors. In *ISCA*, June 2009.

[2] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI*, June 2008.

[3] S. Burckhardt, R. Alur, and M. M. K. Martin. CheckFence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *PLDI*, June 2007.

[4] J. Burnim, K. Sen, and C. Stergiou. Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In *TACAS*, July 2011.

[5] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk Enforcement of Sequential Consistency. In *ISCA*, June 2007.

[6] Y. Duan, X. Feng, L. Wang, C. Zhang, and P.-C. Yew. Detecting and Eliminating Potential Violations of Sequential Consistency for Concurrent C/C++ Programs. In *CGO*, March 2009.

[7] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, June 1998.

[8] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *ICPP*, August 1991.

[9] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA*, June 1999.

[10] Intel Corp. *IA-32 Intel Architecture Software Developer Manual, Volume 2: Instruction Set Reference*. 2002.

[11] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Tran. on Comp.*, July 1979.

[12] J. Lee and D. Padua. Hiding Relaxed Memory Consistency with Compilers. In *PACT*, October 2000.

[13] C. Lin, V. Nagarajan, and R. Gupta. Efficient Sequential Consistency using Conditional Fences. In *PACT*, September 2010.

[14] C. Lin, V. Nagarajan, R. Gupta, and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *ASPLOS*, March 2012.

[15] D. Marino, A. Singh, T. Millstein, M. Musuvathi, and S. Narayanasamy. A Case for an SC-preserving Compiler. In *PLDI*, June 2011.

[16] J. F. Martinez and J. Torrellas. Speculative Synchronization: Applying Thread-Level Speculation to Explicitly-Parallel Applications. In *ASPLOS*, October 2002.

[17] A. Muzahid, S. Qi, and J. Torrellas. Vulcan: Hardware Support for Detecting Sequential Consistency Violations Dynamically. In *MICRO*, December 2012.

[18] X. Qian, B. Sahelices, J. Torrellas, and D. Qian. Volition: Scalable and Precise Sequential Consistency Violation Detection. In *ASPLOS*, March 2013.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *MICRO*, December 2001.

[20] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap Between Memory Consistency Models. In *SPAA*, June 1997.

[21] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos. SESC Simulator, January 2005. http://sesc.sourceforge.net.

[22] P. Sewell, S. Sarkar, M. O. Myreen, and S. Owens. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *CACM*, May 2010.

[23] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM TOPLAS*, April 1988.

[24] A. Singh, S. Narayanasamy, D. Marino, T. D. Millstein, and M. Musuvathi. End-to-End Sequential Consistency. In *ISCA*, June 2012.

[25] SPARC International, Inc. *The SPARC Architecture Manual (Version 9)*. 1994.

[26] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *PPoPP*, June 2005.

[27] T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Mechanisms for Store-wait-free Multiprocessors. In *ISCA*, June 2007.